Efficient Versioning for Scientific Array Databases

Efficient Versioning for Scientific Array Databases

Adam Seering[1], Philippe Cudre-Mauroux[1,2], Samuel Madden[1], and Michael Stonebraker[1]

[1]*MIT CSAIL – USA*
aseering@mit.edu, {pcm,madden,stonebraker}@csail.mit.edu

[2]*U. of Fribourg – Switzerland*
pcm@unifr.ch

***Abstract*— In this paper, we describe a versioned database storage manager we are developing for the SciDB scientific database. The system is designed to efficiently store and retrieve array-oriented data, exposing a "no-overwrite" storage model in which each update creates a new "version" of an array. This makes it possible to perform comparisons of versions produced at different times or by different algorithms, and to create complex chains and trees of versions.**

We present algorithms to efficiently encode these versions, minimizing storage space or IO cost while still providing efficient access to the data. Additionally, we present an optimal algorithm that, given a long sequence of versions, determines which versions to encode in terms of each other (using delta compression) to minimize total storage space. We compare the performance of these algorithms on real world data sets from the National Oceanic and Atmospheric Administration (NOAA), OpenStreetMaps, and several other sources. We show that our algorithms provide better performance than existing version control systems not optimized for array data, both in terms of storage size and access time, and that our delta-compression algorithms are able to substantially reduce the total storage space when versions exist with a high degree of similarity.

I. Introduction

In the SciDB project (`http://scidb.org`), we are building a new database system designed to manage very large array-oriented data, which arises in many scientific applications. Rather than trying to represent such arrays inside of a relational model (which we found to be inefficient in our previous work [1]), the key idea in SciDB is to build a database from the ground-up using arrays as the primary storage representation, with a query language for manipulating those arrays. Such an array-oriented data model and query language is useful in many scientific applications, such as astronomy and biology settings, where the raw data consists of large collections of imagery or sequence data that needs to be filtered, subsetted, and processed.

As a part of the SciDB project, we have spent a large amount of time talking to scientists about their requirements from a data management system (see the "Use Cases" section of the `scidb.org` website), and one of the features that is consistently cited is the need to be able to access historical versions of data, representing, for example, previous sensor readings, or derived data from historical raw data (and implying the need for a no overwrite storage model.)

In this paper, we present the design of the no-overwrite storage manager we have developed for SciDB based on the concept of *named versions*. Versions allow a scientist to engage in "what-if" analyses. Consider, for example, an astronomer with a collection of raw telescope imagery. Imagery must be processed by a "cooking" algorithm that identifies and classifies celestial objects of interest and rejects sensor noise (which, in digital imagery, often appears as bright pixels on a dark background, and is quite easy to confuse for a star!) An astronomer might want to use a different cooking algorithm on a particular study area to focus on his objects of interest. Further cooking could well be in order, depending on the result of the initial processing. Hence, there may be a tree of versions resulting from the same raw data, and it would be helpful for a DBMS to keep track of the relationships between these objects.

In this paper, we focus on the problem of how to best store a time-oriented collection of versions of a single large array on disk, with the goal of minimizing the storage overhead (devices like space telescopes can produce terabytes of data a day, often representing successive versions of the same portion of the sky) and the time to access the most commonly read historical versions. Our system supports a tree of named versions branching off from a single ancestor, and provides queries that read a "slice" (a hyper-rectangle) of one or a collection of versions. Our system includes algorithms to minimize the storage space requirements of a series of versions by encoding arrays as deltas off of other arrays.

Specifically, our contributions include:

- New optimal and approximate algorithms for choosing which versions to materialize, given the distribution of access frequencies to historical versions.
- Chunking mechanisms to reduce the time taken to retrieve frequently co-accessed array portions.
- Experiments with our system on a range of dense and sparse arrays, including weather simulation data from NOAA, renderings of road maps from OpenStreetMaps, video imagery, and a linguistic term-document frequency matrix.
- Comparisons of our system to conventional version-control systems, which attempt to efficiently store chains of arbitrary binary data, but do not include sophisticated optimization algorithms or array-specific delta operations. We show that our algorithms are able to use 8 times less space and provide up to 45 times faster access to stored data.
- Comparisons of the performance of a variety of different delta algorithms for minimizing the storage space of a collection of versions from our scientific data sets.

The results we present come from a stand-alone prototype, which we used to refine our ideas and architecture. We are currently adding our code to the open-source SciDB production system. In the rest of this document, we describe the architecture and interface of the system we have built in Section II. We detail our delta-algorithm for differencing two or more arrays in Section III and our optimization algorithms for choosing versions to materialize in Section IV. Finally, we present the performance of our system in Section V, describe related work in Section VI, and conclude in Section VII.

II. System Overview

The basic architecture of our versioning system is shown in Figure 1. Our goal is to prototype options for SciDB; hence, our storage system mimics as much as possible the SciDB system. In fact, we are now adding the functionality described herein to the open source and publicly available code line (see Appendix A for examples of how the SciDB query system is being extended to support access to versions.)

The query processor receives a declarative query or update from a front end that references a specific version(s) of a named array or arrays. The query processor translates this command into a collection of commands to update or query specific versions in the storage system.

Each array may be partitioned across several storage system nodes, and each machine runs its own instance of the storage system. Each node thereby separately encodes the versions of each partition on its local storage system. In this paper we focus on the storage system for a single-node, since each node will be doing identical operations. The reader is referred to other SciDB research on the partitioning of data across nodes [2].

Our storage system deals with named arrays, each of which may have a tree of versions, uniquely identified by an id. We support five basic array and version operations: allocate a new array, delete an array, create a new version of an array, delete a version of an array, and query a version of an array.

Note that our prototype is a *no-overwrite* storage system. Hence, it is never possible to update an existing version once it has been created – instead, all updates are performed by creating a new version in the system. The goal of the versioning system is to efficiently encode updates so as to minimize the space required on disk.

In the rest of this section, we describe the two major access methods for the versioning system: new version insertion, and query for a version.

A. Version Manipulation Operations

Before a version can be added, the query processor must issue a *Create* command which initializes an array with a specific name and schema. Arrays have typed and fixed-sized *dimensions* (e.g., X and Y are integer coordinates ranging between 0 and 100) that define the cells of the array, as well as typed *attributes* that define the data stored in each cell (e.g., temperature and humidity are floating point values).

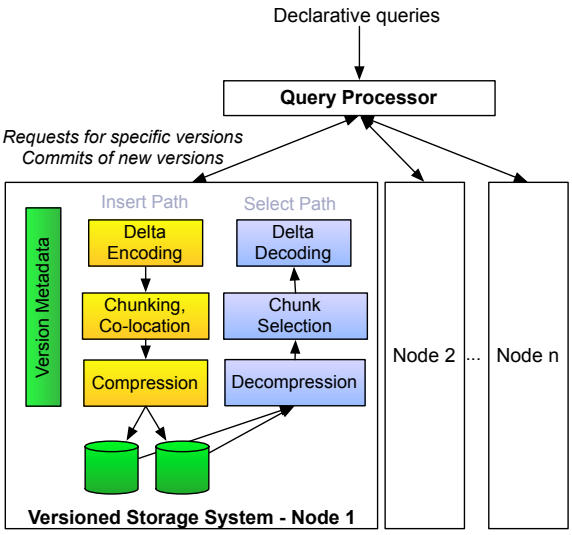

Fig. 1. SciDB version system architecture, showing steps for inserting a new version and selecting a version.

To add a version, the query processor uses the *Insert* operation, which supplies the contents of the new version, and the name of the array to append to, and then adds it as a new version to the database. The payload of the insert operation takes one of three forms:

1) A *dense representation*, where the value of every attribute in every cell is explicitly specified in a row major order. In this case, the dimension values can be omitted.

2) A *sparse representation*, in which a list of (*dimension*, *attribute*) value pairs are supplied, along with a default-value which is used to populate the attribute values for unspecified dimension values.

3) A *delta-list representation*, in which a list of (*dimension*, *attribute*) value pairs are supplied along with a base-version from which the new version inherits. The new version has identical contents to the old version except in the specified dimension values.

The versioning system also supports a *Branch* operation that accepts an explicit parent array as an argument, which must already exist in the database. Branch operates identically to Insert except that a new named version is created instead of a new temporal version of the existing array.

Finally, the versioning system also includes a *Merge* operation that is the inverse of Branch. It takes a collection of two or more parent versions and combines them into a new sequence of arrays (it does not attempt to combine data from two arrays into one array.) Note that the existence of merge allows the version hierarchy to be a graph and not strictly a tree.

An insert or branch command is processed in three steps, as shown in Figure 1. First, the payload is analyzed so it can be encoded as a delta off of an existing version. Delta-ing is performed automatically by comparing the new version to versions already in the system, and the user is not required to load the version using the delta-list representation to take advantage of this feature.

Second, the new version is "chunked", meaning that it is split into a collection of storage containers, by defining a fixed

stride in each of the dimensions. Fixed-size chunks have been shown in [2] to have the best query performance and have been adopted both in our prototype and in SciDB.

Third, each chunk is optionally compressed and written to disk. Data is added to the *Version Metadata* indicating the location on disk of each chunk in the new version, as well as the coordinates of the chunks and the timestamp of the version, as well as the versions against which this new version was delta'ed (if any). Since chunks have a regular structure, there is a straight-forward mapping of chunk locations to disk containers, and no indexing is required.

We describe the details of our delta-encoding and delta-version query algorithms in Section III and Section IV.

B. Version Selection and Querying

Our query processor implements four *Select* primitives. In its first form, it takes an array name and a version ID, and returns the contents of the specified version. In its second form, it takes an array name, a version ID, and two coordinates in the array representing two opposite corners of a hyper-rectangle within the specified array.

In its third form, select accepts an array name and an ordered list of version IDs. Given that the specified arrays are N-dimensional, it returns an $N+1$-dimensional array that is effectively a stack of the specified versions. So, for example, if array A were returned, $A(1,:)$ would be the first version selected; $A(2,:)$ would be the second version selected, etc.

The fourth form is a combination of the previous two: It takes an array name, an ordered list of version IDs, and two coordinates specifying a hyper-rectangle. It queries the specified ranges from each of the specified versions, then stacks the resulting arrays into a single $N+1$-dimensional array and returns it.

These operations are handled by a series of processing steps. First, the chunks that are needed to answer the query are looked up in the Version Metadata. Since each version chunk is likely delta-ed off of another version, in general, a chain of versions must be accessed, starting from one that is stored in native form. This process is illustrated in Figure 2. Here, there are three versions of an array, each of which are stored as four chunks. Version 3 is delta-ed against Version 2, which is delta-ed against Version 1. The user performs a query asking for a rectangular region in Version 3. To access this data, the system needs to read 6 chunks, corresponding to the chunks that overlap the queried region in all three versions.

The required chunks are read from disk and decompressed. If the accessed version is delta-ed against other versions, the delta-ing must be unwound. Finally, a new array containing the result of the query is generated in memory and returned to the user.

C. Querying Schema and Metadata

Besides operations to create and delete versions, the versioning system supports the ability to query metadata. It includes a *List* operation, that returns the name of each array currently stored in the system. Second, it supports a *Get Versions* operation, that accepts an array name as an argument, and returns an ordered list of all versions in that array. It also provides facilities to look up versions that exist at a specific date and time, and methods to retrieve properties (e.g., size, sparsity, etc.) of the arrays.

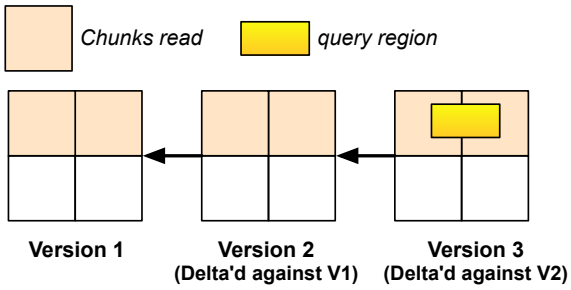

Fig. 2. Diagram illustrating a chain of versions, with a query over one of the versions. The answer the query, the 6 highlighted chunks must be read from disk.

D. Integration into SciDB Prototype

We are now in the process of integrating our versioning system into the SciDB engine. In essence, we are gluing the SciDB query engine onto the top of the interfaces described in the previous two sections. Detailed syntax examples are given in Appendix A.

III. Storing and Compressing Arrays

In this section we introduce a few definitions and describe how arrays are stored and compressed on disk in our prototype.

A. Definitions

An array A_i is a bounded rectangular structure of arbitrary dimensionality. Each combination of dimension values defines a cell in the array. Every cell either holds a data value of a fixed data type, or a $NULL$ value. The cells of a given array are homogeneous, in the sense that they all hold the same data type. We say that an array is *dense* if all its cells hold a data value, and is *sparse* otherwise.

The versioning problem arises in settings where application-specific processes (e.g., data collections from sensors, scientific simulations) iteratively produce sequences of related arrays, or *versions* that each captures one state of some observed or simulated phenomenon over time. The other way the versioning problem arises is when an array is subject to update, for example to correct errors. In either case, we denote the versions $0 \ldots T$ of array A_i as $A_i^0, \ldots, A_i^T$.

B. Storing Array Versions

There exist many different ways of storing array versions. The most common way is to store and compress each array version independently (the sources we took our data from all use this technique to store their arrays, see Section V). However, if consecutive array versions are similar, it may be advantageous to store the differences between the versions instead of repeating all common values. We describe below the various storage mechanisms we consider to store array versions.

1) Full Materialization: The simplest way to store a series of array versions $A^0, \ldots, A^T$ is to store each version separately. For dense arrays, we store all the cell values belonging to a given version contiguously without any prefix or header to minimize the storage space. For sparse arrays, we have two options: either we store the values as a dense array and write a special *NULL* value for the missing cells, or we write a series of values preceded by their position in the array and store the *NULL* values implicitly.

Recall that arrays are "chunked" into fixed sized sub-arrays. The size of an uncompressed chunk (in bytes) is defined by a compile-time parameter in the storage system; by default we use 10 Mbyte chunks (see Section V-B). The storage manager computes the number of cells that can fit into a single chunk, and divides the dimensions evenly amongst chunks. For example, in a 2D array with 8 byte cells and 1 Mbyte chunks, the system would store 1 Mbyte / 8 bytes = 128 kcells/chunk. Hence each chunk would have dimensionality $dim = \lceil \sqrt{128K} \rceil = 358$ units on a side. Each chunk of each array is stored in a separate file in the file system, named by the range of cells in the chunk (e.g., `chunk-0-0-357-357.dat`, `chunk-0-357-357-714.dat`, `...`). To look up the file containing cell X, Y the system computes

$$fX = \lfloor X/dim \rfloor \times dim, fY = \lfloor Y/dim \rfloor \times dim$$

and reads the file `chunk-fX-fY-fX+1-fY+1.dat`.

Every version of a given array is chunked identically. In our prototype, compressing and delta-ing chunks is done on a chunk-by-chunk basis (in the SciDB project as a whole we are exploring more flexible chunking schemes.)

2) Chunk Compression: Our system is able to compress individual versions using popular compression schemes. We took advantage of the SciDB compression library [3] to efficiently compress individual chunks. This library includes various compression schemes such as Run-Length encoding, Null Suppression, and Lempel-Ziv compression. Additionally, we added compression methods based on the JPEG2000 and PNG compressors, which were developed explicitly for images.

3) Delta Encoding: A *delta* is simply the cell-wise difference between two versions: Given two versions A^i and A^j, each cell value in A^j is subtracted from the corresponding cell value in A^i to produce the delta. Deltas can only be created between arrays of the same dimensionality. In this section, we discuss how we compute deltas; the problem of choosing which arrays to actually delta against each other is more complex and is the subject of Section IV.

If versions A^i and A^j are similar, their delta will tend to be small. As a result, the delta can be stored using fewer bits per cell than either A^i or A^j. Our algorithm stores the delta as a dense collection of values of length D bits. We compute the smallest value of D that can encode all of the deltas in $A^i[n] - A^j[n]$. We write the delta array δ_A^{ij} to disk, such that cell n of this array contains $A^i[n] - A^j[n]$.

As an additional optimization, if more than a fraction F of cells in δ_A^{ij} can be encoded using $D' > D$ bits per cell, we create a separate matrix and store cells that require D' bits per cell separately (either as a sparse or dense array, depending on which is better.)

The system also supports bit depths of 0, and empty sparse arrays. Hence, if A^i and A^j are identical, the delta data will use negligible space on disk. Furthermore, if an array would use less space on disk if stored without delta compression, the system will choose not to use it. Disk space usage is calculated by trying both methods and choosing the more economical one.

Finally, we implemented two different ways of storing the deltas on disk: the first method stores all the deltas belonging to a given version together in one file, while the second method co-locates chains of deltas belonging to different versions but all corresponding to the same chunk. Unless stated otherwise, we consider co-located chains of deltas in the following, since they are more efficient.

IV. Version Materialization

In the following, we propose an efficient algorithms to decide how to encode the versions of an array. The challenge in doing this is that we have to choose whether or not to *materialize* (e.g., physically store) each array, or to delta it against some other array. If we consider a series of n versions, we have for each version n possible choices for its encoding (since we either materialize the version or delta it against one of the $n-1$ other versions), and (in the worst case) the optimal choice will depend on how every other version was encoded; hence a naive algorithm may end up considering n^n possible materialization choices.

For cases where the workload is heavily biased towards the latest version, the optimal algorithm boils downt to materializing the latest version and to delta all previous ones. While this is a frequent case in practice, other workloads (returning for example arbitrary ranges of versions, or returning small portions of arbitrary versions) are from our experience also very frequent in science. In many cases in astronomy or remote sensing, for instance, following objects in time and space requires to perform subqueries returning subregions of the arrays for relatively long ranges of versions.

We describe below an efficient algorithm to determine optimal encodings for an arbitrary number of versions and arbitrary workloads, without exploring all materialization options. Before running the algorithms described below, we first compute whether, from a space-perspective, delta-compression is valuable for this array. We do this using a sample series of consecutive versions $(A^1, \ldots, A^n)$. We also decide if the array should be stored using a dense or a sparse representation and what compression algorithm should be used.

If delta compression is determined to be effective, we perform a search for the optimal delta encoding as described below.

A. Materialization Matrix

The algorithm starts by constructing a data structure, called the *Materialization Matrix*, to determine the value of delta-encoding for the versions at hand. The Materialization Matrix

MM is a $n \times n$ matrix derived from series of versions (all versions are of the same dimensionality). The values $MM(i,i)$ on the diagonal of this matrix give the space required to materialize a given version V^i: $MM(i,i) \equiv M^i$. The values off the diagonal $MM(i,j)$ represent the space taken by a delta $\Delta^{i,j}$ between two versions V^i and V^j. Note that this matrix is symmetric. This matrix can be constructed in $O(n^2)$ pairwise comparisons.

We have implemented several ways of efficiently constructing this matrix. In particular, we have found that computing the space S to store the deltas based on a random sample of R of the total of N cells for a pair of matrices and then computing $S \times R/N$ yields a fairly approximate estimate of the actual delta size, even for S/N values of .1% or less. We are also exploring the use of transformations (e.g., using harmonic analyses) of large versions in order to work on smaller representations.

B. Layouts

After having computed the materialization matrix, our versioning system determines the most compact delta encoding for the series of versions. For each V^i in a collection of n versions $(V^1, \ldots, V^n)$ we can:

1) materialize the compressed representation, which requires $MM(i,i)$ bytes, or

2) delta V^i against any other version, requiring $\Delta^{i,j}$ bytes.

Thus, there are n different ways of storing each version. However, not all encodings lead to collections of versions that can be retrieved by the system. Suppose for example that we encode three versions V^1, V^2, V^3 through three deltas—storing V^1 as a delta $\Delta^{1,2}$ against V^2, V^2 as a delta $\Delta^{2,3}$ against V^3, and V^3 as a delta $\Delta^{3,1}$ against V^1. In this case, none of the versions can be reconstructed because the sequence of deltas forms a loop. To be able to reconstruct a version, one must either materialize that version, or link it through a series of deltas to a materialized version. We call an encoding of a collection of versions where all versions can be reconstructed a *valid layout*.

In the following, we use a graph representation to model the way versions are stored. In this representation, a layout is a collection of nodes (representing versions) and edges (representing delta encoding or materialized versions). A materialized version is represented as an arc with the same source and destination node. If a version is delta encoded, then it is represented as an arc from the source to the destination. We do not consider replication of versions, so each node has exactly one incoming arc. Figure 3 gives an illustration of a valid and an invalid layout. We draw a few observations from those definitions (we omit the simplest proofs due to space constraints):

Observation 1: A layout of n versions always contains n edges.

Observation 2: A layout containing at least one cycle of length > 1 is invalid.

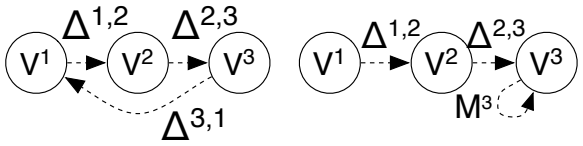

Fig. 3. Two ways of storing three versions $(V^1, \ldots, V^3)$; The first (left) is invalid since the series of deltas creates a cycle; The second (right) is valid since every version can be retrieved by following series of deltas from V^3, which is materialized .

(Note that we use an undirected version of the graphs when detecting cycles, since our system can reconstruct the versions in both directions, by adding or subtracting the delta).

Proof: Consider a layout containing a cycle connecting C versions (with $1 < C \leq n$). This cycle contains C edges, and a total of $n - C$ nodes and edges remain outside the cycle. At least one edge must be used to materialize one version (a layout without any materialization is never valid). If one of the versions along the cycle is materialized, then $n - C - 1$ edges remain to connect $n - C$ versions, which can never produce a valid layout (since each node must have one outgoing edge). If none of the versions along the cycle is materialized, then at least one version outside of the cycle must be materialized. Let us assume that M versions (with $1 < M \leq n - C$) are materialized. $n - C - M$ versions must then be connected directly or indirectly to the materialized nodes, taking $n - C - M$ edges and leaving the nodes belonging to the cycle invalid.

Observation 3: a layout containing a set of connected components where each component has one and exactly one materialized version is valid.

Observation 4: a layout without any (undirected) cycle of length > 1 is always valid; hence, without considering the materialization edges, a valid layout graph is actually a *polytree*.

Proof: The layout contains n edges. There is at least one materialized version in this case (a *tree* connecting all the nodes takes $n - 1$ edges; any additional edge creates a loop). Let us assume that there are M materialized versions. Thus, there are $n - M$ edges interconnecting the n nodes. Since there is no cycle, those $n - M$ edges and n nodes form a *forest* (i.e., a disjoint union of trees) composed of M trees. Each of the trees contains exactly one materialized version (since we have exactly one outgoing edge per node), and hence the layout is valid.

C. Space Optimal Layouts

We now describe how to efficiently determine the most compact representation of a collection of versions by using the Materialization Matrix. We first consider the case where we assume materializing a version always takes more space than delta'ing it from another version (i.e., $MM(i,i) > MM(i,j)\ \forall j \neq i$), which should be true in most cases, and then briefly discuss how to relax this assumption. When this assumption is true, the minimal layout is composed of n deltas

connected to one materialized version. The optimal layout can in this case be derived from the *minimum spanning tree* of the collection of versions.

[Sketch of] Proof: since materializations are always more expensive than deltas, the minimal layout must be composed of a minimum number of materializations, namely one, and a series of deltas interlinking all versions. The delta arcs must form a tree since loops are not allowed (see above). By definition, the cheapest subgraph connecting all the vertices together is the minimum weight spanning tree, where the weight of an edge between vertex V_i encoded in terms of vertex V_j is $\Delta^{i,j}$.

Algorithm 1 finds the space-optimal layout by considering an undirected materialization graph and its minimum spanning tree. The output Λ contains an encoding strategy for every version (either materializing it or encoding it in terms of some other version.) The minimum spanning tree can be found in linear time using a randomized algorithm [4].

```
/* Create The Undirected Materialization
   Graph                                          */
Create Graph G
foreach version V^i ∈ {V^1, ..., V^N} do
    Add Node V^i, G
foreach version V^i ∈ {V^1, ..., V^N} do
    foreach MM(i, j) | j < i do
        Add Edge V^i, V^j, MM(i, j), G
/* Find Minimal Spanning Tree on Undirected
   Graph                                          */
MST = MinimumSpanningTree(G)
/* Take Cheapest Materialization As Root          */
V^i_min = min(MM(i, i)) ∀ 1 ≤ i ≤ N
Λ = V^i_min
/* Traverse MST and Add Deltas from Root          */
foreach Pair of Nodes (V^i, V^j) ∈ MST from root V^i_min do
    Λ = Λ ⊕ Δ^{i,j}
/* store the versions as given by best
   layout                                         */
store Λ
```

Algorithm 1: Algorithm for finding the minimum storage layout.

If there is a materialized version that is cheaper than some delta, then the above algorithm might fail. In particular, materializing more than one version might result in a more compact layout than the one given by the above minimum spanning tree of deltas. Specifically, a more compact representation of the versions could actually be a minimum spanning forest with more than one materialization.

We generalize our algorithm to deal with this situation. We first run the above algorithm to find the most compact representation. Then, we continue with an examination of each version that might be stored more compactly through materialization. If there exists a delta on the path from that version to the root of the tree that is more expensive than the materialization, then it is advantageous to split the graph by materializing that version instead of considering it as a delta. The complete algorithm is given in Appendix B.

D. Workload-Aware Layouts

In certain situations, scientists have some a priori knowledge about the workload used to query the versions, either because they have some specific algorithm or repeated processing in mind (e.g., always comparing the last ten versions) or because they have a sample historical workload at their disposal. Below, we describe algorithms to determine interesting layouts that take advantage of this workload knowledge.

Knowledge about the workload allows us to minimize the total I/O, measured in terms of the number of disk operations (i.e., reads and writes), rather than simply minimizing the number of bytes on disk as above. Because chunks read from disk in SciDB are relatively large (i.e., several megabytes), disk seeks are amortized so that we can count use the number of chunks accessed as a proxy for total I/O cost. This further suggests that chunks that are frequently queried should be stored compactly on disk in order to minimize the total number of bytes read when executing queries.

To model the cost of a query q (which can either represent a snapshot or a range query), we first determine the set of versions $\mathcal{V}_\Lambda(q)$ that have to be retrieved to answer q; this set is composed of the union of all versions directly accessed by the query, plus all further versions that have to be retrieved in order to reconstruct the accessed versions (corresponding to all versions that can be reached by following edges from the set of accessed versions in the materialization graph). The cost of a query q can then be expressed as $Cost_\Lambda(q) \sim \sum_{V^i \in \mathcal{V}_\Lambda(q)} Size_\Lambda(V_i)$ (where $Size()$ returns the size in KB of the corresponding version). Considering a workload Q composed of frequent queries along with their respective weights (i.e., frequencies) w, the optimal layout minimizing I/O operations for the workload is:

$$\Lambda_Q = \underset{\Lambda}{\operatorname{argmin}} \left(\sum_{q_j \in Q} w_j * Cost_\Lambda(q_j) \right).$$

We ignore caching effects above for simplicity, and since they are often negligible in our context for very large arrays or complex workloads.

Layouts yielding low I/O costs will typically materialize versions that are frequently accessed and hence won't be optimal in terms of storage, so we can't simply use the algorithms described above. Instead, we need to: 1) find all spanning trees (or forests if we consider the algorithm from Appendix B) derived from the Materialization Matrix – in our case, the number of possible spanning trees for a complete undirected graph derived from a Materialization Matrix is given by Cayley's formula [5] and is exponential with the number of versions n; 2) consider all possible ways of materializing each tree (second part of Algorithm 1), and 3) compute for each query and each valid layout the set of versions $\mathcal{V}_\Lambda$ that have to be retrieved from disk – this operation is complex, since each layout might lead to a different set of versions to retrieve for a given query.

As an alternative, we describe an efficient heuristic algorithm to determine layouts yielding low I/O costs below. For a workload composed of isolated snapshots and range queries,

one can consider each query independently and construct the I/O optimal layout by determining the space-optimal layout for each range/snapshot in isolation (for snapshots, the solution is straightforward and boils down to materializing all queried versions) in order to minimize the total cost expressed above. For overlapping queries, things are more complex. Given two queries retrieving two overlapping lists of versions $\boldsymbol{V}_1$ and $\boldsymbol{V}_2$, one has to enumerate all potential layouts as described above on $\boldsymbol{V}_1 \cup \boldsymbol{V}_2$ to find the I/O optimal layout. In practice, however, only four layouts are interesting in this case: i) a layout where all $\boldsymbol{V}_1 \setminus \boldsymbol{V}_2$, $\boldsymbol{V}_2 \setminus \boldsymbol{V}_1$ and $\boldsymbol{V}_1 \cap \boldsymbol{V}_2$ are stored most compactly for workloads where both $\boldsymbol{V}_1$ and $\boldsymbol{V}_2$ are frequently accessed ii) a layout storing both $\boldsymbol{V}_1$ and $\boldsymbol{V}_2 \setminus \boldsymbol{V}_1$ in their most compact forms for workloads where $\boldsymbol{V}_1$ is accessed more frequently, iii) same as i) with $\boldsymbol{V}_1$ and $\boldsymbol{V}_2$ inverted for workloads where $\boldsymbol{V}_2$ is accessed more frequently, and finally iv) a layout where $\boldsymbol{V}_1 \cup \boldsymbol{V}_2$ is stored more compactly—which might be interesting for settings where both $\boldsymbol{V}_1$ and $\boldsymbol{V}_2$ are frequently accessed and where materializations are very expensive. This divide and conquer algorithm can be generalized for N overlapping queries delineating M segments, by considering the most compact representation of each segment initially, and by combining adjacent segments iteratively.

E. Handling New Versions

When a new version is added, we do no want to immediately re-encode all previous versions. The simplest option is to update the materialization matrix, and use it to compute the best encoding of the new version in terms of previous versions. Periodically we may wish to recompute the optimal version tree as a background re-organization step. As a more sophisticated update heuristic, we can accumulate a batch of K new versions, and compute the optimal encoding of them together (in terms only of the other versions in the batch) using the algorithms described above. This batch will have one or more materialized arrays in it. If we wish to avoid storing this materialized array, we can compare the cost of encoding it in terms of the materialized arrays in other (previously inserted) batches. In practice, however, we have found that as long as K is relatively large (say 10–100), it is sufficient to simply keep these batches separate. This also has the effect of constraining the materialization matrix size and improving query performance by avoiding very long delta chains.

Finally, it is important to point out that in many practical contexts, the workload is heavily biased towards the latest version of the array (e.g., for scientists who only query the current version of a given array, but who still want to maintain its history on disk for traceability purposes.) In that case, the optimal algorithm is actually simpler: the newest version is always materialized since it is heavily queried. All the other versions are then stored in the most compact way possible, either by using the spanning-tree algorithm described above, or by iteratively inspecting each version and deciding whether delta-ing it against the new version would yield a more compact representation than the current one.

V. Performance Evaluation

In this section, we evaluate the array system and version materialization algorithms described in the previous sections on a number of data sets. The objective of this section is to understand how our versioning system compares to other, more general-purpose systems like Subversion (SVN) and Git, to measure its overall compression and query performance, and to understand when our optimal materialization algorithm approach outperforms a simple linear delta chain. Our experiments are run on a single-node prototype; as noted in the introduction, in the real SciDB deployment each of many nodes will store versions of the pieces of very large arrays assigned to them.

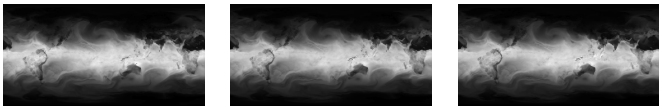

Fig. 4. Three consecutive arrays from the NOAA weather data, measuring Specific Humidity on March 1, 2010. Measurements are scaled to an 8-bit grayscale color space for display. Note that the images are very similar, but not quite identical; for example, many of the sharp edges in the images have scattered single-pixel variations.

We experimented with four data sets. The first data set is a a dense collection of 1,365 approximately 1 MB weather satellite images captured in 15 minute intervals from the US National Oceanic and Atmospheric Administration[1] (NOAA). We downloaded data from their "RTMA" data set, which contains sensor data measuring a variety of conditions that govern the weather, such as wind speed, surface pressure, or humidity, for a grid of location covering the continental United States. Each type of measurement was stored as floating-point numbers, in its own versioned matrix. We took data captured on August 30 and 31 2010 and sampled 15 minutes, for a total of 1,365 matrices. Sample data from this data set is shown in Figure 4.

The second data set is a collection of sparse arrays taken from the Open Mind Common Sense ConceptNet[2] network at MIT. This network is a highly sparse square matrix storing degrees of relationships between various "concepts". Only the latest ConceptNet data is accessed regularly, but their server internally keeps snapshots for backups; the benchmark data consisted of all weekly snapshots from 2008. Each version is about 1,000,000 by 1,000,000 large with around 430,000 data points (represented as 32-bit integers).

The third data set consists of a collection of 16 large (1 GB) dense array from Open Street Maps[3] (OSM)—a free and editable collection of maps. We selected tiles from the region overlooking Boston at zoom level 15 (from GPS coordinates [-72.06, 41.76] to [-70.74, 42.70]). We selected 16 consecutive versions for our experiments, one per week for the last 16 week of 2009.

[1] *http://www.noaa.gov/*
[2] *http://csc.media.mit.edu/conceptnet*
[3] *http://www.openstreetmap.org/*

The fourth data set, finally, is a dense, periodic data set. It was taken from the Switch Panorama archive[4]. We used every 80th view taken from Zürich's observatory for one week, ranging from February 14 through February 20, 2011. The observatory recorded and published 2,003 views within that timeframe. We selected these data sets to exercise different aspects of the system; e.g., dense (NOAA and OSM) vs. sparse storage (Open Mind), and finding non-trivial delta materialization strategies (Switch Panorama). The OSM data generally differs less between consecutive versions (and is thus more amenable to delta compression) than the NOAA data, because the street map evolves less quickly than weather does.

Our results were obtained on a Intel Xeon quad-core (Intel E7310) running at 1.6 GHz, with 8 GB of Ram and a 7200RPM SATA drive running Ubuntu Lucid. Our prototype is written in a mix of Python 2.6 with the delta and compression libraries coded in C (note that SciDB itself is entirely coded in C, so we are in the process of porting all of our code to C.)

A. Delta Encoding and Compression

In our first experiment, we compare the algorithms we developed for delta encoding two versions as well as the compression methods we implemented, as described in Section III.

Delta Experiments: We begin with our delta algorithms. The algorithms differ in how they attempt to reduce the size of array deltas. They all begin with an array representing the arithmetic cellwise difference between two arrays to be delta encoded. The "dense" method reduces the number of bytes used to store the array as much as possible without losing data, under the assumption that each difference value will tend to be small. The "sparse" method is a no-op for sparse arrays; for dense arrays, it converts the difference array into a sparse array, under the assumption that relatively few differences will have nonzero values. The "hybrid" method calculates an optimal threshold value and splits the delta array into two arrays, one (sparse or dense) array of large values and one (dense) array of small values, as described in Section III-B.3. The MPEG-2-like matcher is built on top of hybrid compression, but the target array is broken up into 16x16 chunks and each chunk is compared to every possible region in a 16-cell radius around its origin, in case the image has shifted in one direction.

We ran these algorithms on the first 10 versions of the NOAA data set. This data set contains multiple arrays at each version, so there were a total of 88 array objects. The results are shown in Table I. We also experimented with other data sets and found similar results.

For this data, delta compression slows down data access. This is largely because of the computational costs of regenerating the original arrays based on the corresponding two deltas. With the uncompressed array, the raw data is read directly from disk into memory, but with each of the other formats, it is processed and copied several times in memory.

The MPEG-2-like matcher is very expensive as compared to the hybrid algorithm. Its cost is roughly proportional to the number of comparisons it is doing: It is considering all arrays within a region of radius 16 around the initial location. This means $16 \times 16 = 256$ times as many comparisons, for a 2-dimensional array.

[4] *http://cam.switch.ch/cgi-bin/pano2.pl*

TABLE I
PERFORMANCE OF SELECTED DIFFERENCING ALGORITHMS

Delta Algorithm	Import Time	Size	Query Time
Uncompressed	4.31 s	253MB	2.75 s
Dense	8.99 s	168MB	3.41 s
Sparse	21.15 s	191MB	3.21 s
Hybrid	15.16 s	142MB	2.81 s
MPEG-2-like Matcher	9598.10 s	138MB	39.60 s
BSDiff [6]	343.80 s	133MB	3.59 s

TABLE II
COMPRESSION ALGORITHM PERFORMANCE ON DELTA ARRAYS

Compression	Size	Query Time
Hybrid Delta only	133M	3.53s
Lempel-Ziv [7]	94M	4.01s
Run-Length Encoding	133M	3.32s
PNG compression	116M	5.93s
JPEG 2000 compression	118M	20.23s

Of the remaining implementations, the hybrid implementation yields the smallest data size and the best query performance. The improved query performance is most likely caused by the smaller data size: Less time is spent waiting on data to be read from disk, and the required CPU time isn't hugely greater than that of either of the other techniques.

The standalone BSDiff [6] arbitrary-binary-differencing algorithm had the smallest data size overall. However, the algorithm is CPU-intensive, particularly for creating differences, so it had runtimes much longer than those of most of the matrix-based algorithms.

Compression experiments: We compared various compression mechanisms to compress the deltas after they were computed. The results for the NOAA data are shown in Table II. As before, we found similar results with other data sets.

The Lempel-Ziv (LZ) algorithm [7] compresses by accumulating a dictionary of known patterns. Run-length simply stores a list of tuples of the form (value, # of repetitions), to eliminate repeated values. "PNG" and "JPEG 2000" compression are image-compression formats; PNG uses LZ with pre-filtering, and JPEG 2000 uses wavelets. PNG in particular makes heavy use of a variety of tunable heuristics; the particular compression implementation and set of constants used here were those of the Python Imaging Library.

As a comparison, compressing the original array using LZ alone, without first computing deltas, requires 124 MB. Using RLE alone requires 240 MB.

PNG, JPEG 2000 compression, and LZ compression all decreased the data size somewhat. Of those algorithms, LZ had both the smallest resulting data size and the fastest query time of the compression methods, so it is clearly the best overall.

B. Query Performance

In our next experiment, we tested several different versions of our storage manager on the OpenStreetMaps, the NOAA, and

TABLE III
OPEN STREET MAPS, 10 MB CHUNKS, SNAPSHOT QUERY.

Method	1 Array Select		1 Array Subselect	
	Bytes Read	Time	Bytes Read	Time
Chunks + Deltas	1.53 GB	42.63 s	30.20 MB	0.96 s
Chunks	1.00 GB	27.38 s	30.20 MB	1.06 s
Chunks + Deltas + LZ	0.13 GB	18.63 s	2.90 MB	0.61 s
Uncompressed	1.00 GB	192.0 s	1.0 GB	19.65 s

TABLE IV
OPEN STREET MAPS, 10 MB CHUNKS, RANGE QUERY

Method	16 Array Select		16 Array Subselect	
	Bytes Read	Time	Bytes Read	Time
Chunks + Deltas	2.00 GB	249.80 s	42.50 MB	6.86 s
Chunks	15.00 GB	451.01 s	450.00 MB	14.17 s
Chunks + Deltas + LZ	1.89 GB	335.22 s	39.50 MB	10.32 s
Uncompressed	15.00 GB	289.16 s	15.00 GB	276.18 s

the ConceptNet data sets. We begin with our OpenStreetMaps data.

OpenStreetMaps: We start with experiments looking at the effectiveness of chunking on the OpenStreetMap data set, when running queries that select an entire version, several versions, and a small portion of one or several versions. We experimented with various chunk sizes and in the end decided to use 10 MB for all experiments, since it gave good results for most settings. OpenStreetMaps is a good test for this because consecutive versions are quite similar, with just a few changes in the road segments between versions, and so we expected delta compression to work well.

Table III gives the results for *snapshot queries* that retrieve data from the latest versions only. Two different types of queries are run: *full-queries* returning the entire version, and *subqueries* returning only a few cells of a version (i.e., reading only one chunk, approximately 10MB uncompressed). Here, we can see that chunking with delta encoding and LZ compression is generally the best approach, accessing both the least data from disk and providing the lowest query time. In particular, chunking makes subselect operations very fast, and LZ substantially reduces data sizes.

Table IV shows the results for *range queries* that select 16 consecutive versions. Here, LZ leads to the smallest data sets and the least data read, but decompression time for the large arrays in this case turns out to be substantial, causing the uncompressed Chunks + Deltas approach to work better. This is because the large delta chains in this data set do not compress particularly well with LZ.

These results suggest that it might be interesting to adaptively enable LZ compression based on the data set size and the anticipated compression ratios; we leave this to future work.

NOAA and ConceptNet Data: We also experimented with more complex query workloads on the NOAA and ConceptNet (CNet) data, to get a sense of the performance of our approach on other data sets. Table V gives the results for five different workloads: i) Head, where the most recent version is selected with 90% probability, and another single random version is selected with 10% probability (this is repeated 10 times) ii) Random, where a random single version is selected (this is repeated 30 times) iii) Range, where with 10% probability, a random single matrix is selected and with 90% probability, a random range with a standard deviation of 10 is selected (this is repeated 30 times) iv) Mixed, where a query is chosen from the three previous query types with equal probability (this is repeated 15 times) and finally v) Update, where a random modification is made (this is repeated 5 times, each time for a different version chosen uniformly at random). We experimented with arrays compressed using hybrid deltas and Lempel-Ziv compression (H+LZ), with just hybrid deltas (H), and with no compression.

These results show several different effects. First, our delta algorithms, even without LZ, achieve very high compression ratios (3::1 on NOAA, and 35::1 on CNet.) CNet compresses so well because the data is very sparse. Second, in general, compressing the data slows down performance. This is different than in the OSM data because individual versions are small here, and so the reduction in I/O time is not particularly significant, whereas the CPU overhead of decompression is costly (this is particularly true in our Python implementation which makes several in memory copies of arrays when decompressing them.) Third, the performance of the system on compressed sparse data (CNet) is not particularly good, with large overheads incurred when manipulating the data; we believe that this is due to our choice of Python for generating the final result arrays—manipulating and expanding sparse lists is slow.

TABLE V
TESTS ON NOAA AND CONCEPTNET

Data	Comp.	Size	Head	Rand.	Range	Up.	Mix.
NOAA	H+LZ	2.3 GB	11.5 s	16.8 s	17.4 s	2.3 s	104.9 s
	H	2.8 GB	0.3 s	14.8 s	12.9 s	1.0 s	43.3 s
	None	8.2 GB	0.2 s	2.3 s	5.6 s	0.4 s	30.5 s
CNet	H+LZ	11 MB	6.9 s	80.7 s	70.1 s	0.2 s	17.6 s
	H	21 MB	4.8 s	57.0 s	44.5 s	0.2 s	12.6 s
	None	728 MB	0.4 s	2.2 s	3.3 s	0.2 s	1.8 s

C. Comparison to Other Versioning Systems

In this experiment, we compare our system against two widely used general-purpose versioning systems, SVN and GIT. For both SVN and GIT, we mapped each matrix to a versioned file, and committed each version in sequence order. To ensure optimal disk usage, we ran *svnadmin pack* after loading all data into the SVN repository. For GIT, we used *git repack* to compress the data.

We ran our algorithm on the OSM data (which consists of large arrays), and the NOAA data (which consists of smaller arrays). For the OSM data, we looked at queries for the contents of a single whole array, and at a sub-select of just one 10 MB chunk. Results for the OSM data are shown in Table VI. For this data, Git ran out of memory on our test machine. Observe that SVN is substantially slower at loading data, provides less compression (8x), and does not efficiently support sub-selects (because the stored data is not compressed), running about 1.5x slower for whole array reads and 45x slower for single chunk selects. For whole array queries SVN is also somewhat slower, likely because it does not effectively compress (delta) this data.

TABLE VI
SVN AND GIT PERFORMANCE ON OSM DATA

Method	Import Time	Data Size	Array Select	Subselect
Uncompressed	574.5 s	16.0 GB	192.0 s	19.65 s
Hybrid+LZ	2,340.4 s	2.01 GB	18.63 s	0.61 s
SVN	8,070.0 s	16.0 GB	29.2 s	28.6 s
Git	–	–	–	–

TABLE VII
SVN AND GIT PERFORMANCE ON NOAA DATA

Method	Import Time	Data Size	1 Array Select
Uncompressed	4.31 s	253M	2.75 s
Hybrid+LZ	13.1 s	90M	5.47 s
SVN	47.0 s	111M	7.97 s
Git	100.5 s	147M	3.70 s

Results for the NOAA data are shown in Table VII. Here we did not look at array subselects, because each version is only about 1 MB so fits into a single chunk. In this case, Git was successfully able to load the data, although it took much longer than the other systems. For this small data, uncompressed access was the most efficient, because decompression costs on these data sets proved to be relatively high. Hybrid Deltas+LZ, however, yielded the smallest overall data set, and much better load times than SVN or Git.

D. Materialization Experiments

In our final set of experiments, we look at the performance of our materialization algorithm described in Section IV. We compared its performance to a simple linear chain of deltas differenced backwards in time from the most recently added version.

First, we tested our optimal delta algorithm on the Switch dataset, which exhibits some interesting periodicity as adjacent versions (video frames) are very different, but the same scene does occasionally re-occur. Here, our algorithm detects this recurring pattern in the data and computes complex deltas between non-consecutive versions. Our optimal delta algorithm (using hybrid deltas + LZ) compresses the data down to 9.7 MB, while the linear delta-chain algorithm yields a compressed size of 15 MB.

We also tested the performance of the algorithm on a variety of synthetic data sets. These data sets have identical arrays that re-occur every n versions. E.g., for $n = 2$, there are three arrays that occur in the pattern $A1, A2, A3, A1, A2, A3 \ldots$ selected so that each of the n arrays doesn't difference well against the other $n - 1$ arrays. Here, we had 40 arrays, each 8 MB (total size 320 MB with linear deltas); the optimal algorithm for $n = 2$ used 17 MB and for $n = 3$ used 21 MB, finding the correct encoding in both cases.

Loading the delta chain for 40 arrays took 132 s in the optimal case, and 15 s in the linear chain case; most of this overhead is the time to generate the n^2 materialization matrix. Although this is a significant slow down, the load times are still reasonable (about 3s per array), and should be faster when re-implemented in C.

We also confirmed that on a data set where a linear chain is optimal (because consecutive versions are quite similar), our optimal algorithm produces a linear delta chain.

Finally, we ran a series of experiments to test our workload-aware algorithms (see Section IV-D). We ran experiments on our weather data set considering workloads with overlapping range queries (i.e., sets of range queries retrieving 10 images each and overlapping by four versions exactly). The resulting space optimal layouts consider longer delta-chains than the I/O optimal layouts. However, the I/O optimal layout proved to be more efficient when executing the queries. Our system took on average 1.51s to resolve queries on the space optimal layout (results were averaged over 30 runs), while it took only 1.10s on average on the I/O optimal layout, which corresponds to a speedup of 27% in this particular case.

VI. RELATED WORK

In this section, we describe other version control and differencing algorithms.

Version Control Systems: Versioning has a long history in computer science. Naive versioning techniques include forward and backward delta encoding and the use of multiversion B-trees. These techniques have been implemented in various legacy systems including XDFS, Sprite LFS or CVFS [8].

The design of our versioning system is modeled after conventional version-control software such as Subversion and Git. In particular, the concepts of a no-update model and of differencing stored objects against each other for more efficient storage have both been explored extensively by developers of conventional version-control software.

Git in particular is often cited as being faster and more disk-efficient than other similar version-control systems. Significant amounts have been written about its data model [9]: Git stores a version tree and a delta tree, but the two are managed by different software layers and need not relate to each other at all. In order to build an efficient delta tree, Git considers a variety of file characteristics, such as file size and type, in addition to files' relationship in the version tree. It then sorts files by similarity, and differences each file with several of its nearest neighbors to try to find the optimal match. Our approach, in contrast, uses a materialization matrix to efficiently find the best versions to compare against, yielding better theoretical guarantees and better performance in practice than Git, at least on the arrays we used.

Git also ensures that its differences will be read quickly by storing consecutive differences in the same file on disk. This way, if several consecutive differences must be read, they will likely be stored consecutively on disk, eliminating the need for additional disk seeks. Additionally, if a client requests these differences over the network, it will receive several differences (one files' worth) at once, thereby cutting down on the number of required network round-trips. We included a similar *co-location* optimization in SciDB, although we found it did not improve performance significantly.

Image and Video Encoding: Considerable work has been done in the areas of lossy and lossless video encoding, to enable the storage of sequences of 2-dimensional matrices using a minimum of storage space, and while minimizing the

time needed to select an arbitrary matrix from the middle of a sequence. In particular, the HuffYUV lossless codec uses the idea of storing a frame as an approximation plus a small error delta, and the MPEG-1 Part 2 lossy codec selectively materializes arrays in the midst of streams of deltas to improve access times for range selections starting in the middle of the MPEG video stream [10]. These ideas are similar to our idea of materializing some versions in a delta chain, although they do not include an optimal version search algorithm like ours, instead just materializing frames at regular intervals.

Delta encoding is also a widely used technique in video and image compression, with many variants in the literature. For example, video compression codecs like MPEG-1 perform several different types of delta encoding both within and between frames; in this sense they are similar to our encoding techniques [11].

We recently discovered that Gergel *et al.* [12] suggested the use of a spanning tree to encode sets of images a few years ago. However, their model and algorithms are considerably simpler that the ones we developed for our system, since they only consider a single, undirected spanning tree representation (i.e., none of the algorithms described in IV can be captured by their simplistic framework).

Another significant difference between video and image compression techniques and the SciDB versioning systems is that existing techniques support a fixed number of dimensions. Video compression stores a three-dimensional matrix (two dimensions per image and one in time); image compression stores a two-dimensional matrix. Additionally, common implementations of these algorithms tend to assume that this data uses 8-bit unsigned integers for cell values, because 8-bit grayscale (or 8 bits per color channel, with color images) is a common standard for consumer graphics. Some implementations of some of these algorithms support 16-bit unsigned integers; few if any support 32-bit or larger integers, or any other attribute formats.

Versioning and Compression in DBMSs Versioning is similar to features in existing database systems. For example, support for "time travel" is present in many DBMSs, beginning with Postgres [13]. Oracle also includes a feature called "Flash Back"[5] that allows users to run queries as of some time in the past.

Flashback also allows users to create "Flashback Data Archives" that are essentially named versions of tables. Snapshot or backup features like this are present in most databases, and it is common for production database systems to be kept under version control as well. The difference between these tools and our storage manager is that they are not optimized for efficiently encoding versioned array data, as our prototype storage manager is.

Compression has also been considered for databases before [14], [15], [16], [17], [18], but typically the goal is to minimize the size and/or access time of a single relation, rather than the time to access a series of relations as in our case.

[5] http://download.oracle.com/docs/cd/B28359_01/appdev.111/b28424/adfns_flashback.htm

VII. Conclusions

In this paper, we described a prototype versioned storage system we have built for the SciDB scientific database system. Key ideas include efficient, chunk-based storage and compression, and efficient algorithms for determining how to delta-encode a series of versions in terms of each other to minimize storage space or I/O costs. We presented experimental results on a range of real-world imagery and array data, showing that our algorithms are able to dramatically reduce disk space while still providing good query performance. We are currently integrating these ideas into the SciDB system, and adding support to the SciDB declarative query language to provide access to versions. Our code will be available in the next release of SciDB.

Acknowledgements

This work was supported by NSF grants IIS/III-1111371 and SI2-1047955.

References

[1] M. Stonebraker, C. Bear, U. Cetintemel, M. Cherniack, S. Harizopoulos, J. Lifter, J. Rodgers, and S. Zdonik, "One size fits all? part 2: Benchmarking studies," 2007.

[2] E. Soroush, M. Balazinska, and D. Wang, "Arraystore: A storage manager for complex parallel array processing," pp. 253–264, 2011.

[3] P. Cudre-Mauroux, H. Kimura, K.-T. Lim, J. Rogers, R. Simakov, E. Soroush, P. Velikhov, D. L. Wang, M. Balazinska, J. Becla, D. DeWitt, B. Heath, D. Maier, S. Madden, J. Patel, M. Stonebraker, and S. Zdonik, "A demonstration of SciDB: a science-oriented DBMS," *VLDB*, vol. 2, no. 2, pp. 1534–1537, 2009.

[4] D. R. Karger, P. N. Klein, and R. E. Tarjan, "A randomized linear-time algorithm to find minimum spanning trees," *J. ACM*, vol. 42, no. 2, pp. 321–328, 1995.

[5] A. Cayley, "A theorem on trees," *Quart. J. Math*, vol. 23, pp. 376–378, 1889.

[6] C. Percival, "Naive differences of executable code," 2003, unpublished paper.

[7] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE TRANSACTIONS ON INFORMATION THEORY*, vol. 23, no. 3, pp. 337–343, 1977.

[8] N. Zhu, "Data versioning systems," Stony Brook University, Tech. Rep., 2003.

[9] S. Chacon, *Git Community Book*. http://book.git-scm.com/: the Git SCM community, 2010, ch. 1.

[10] Y. Lin, M. S. Kankanhalli, and T. seng Chua, "T.-s.,: Temporal multi-resolution analysis for video segmentation," in *In ACM Multimedia Conference*, 1999, pp. 494–505.

[11] International Standards Organization (ISO), "Coding of moving pictures and audio," Web Site, July 2005, http://mpeg.chiariglione.org/technologies/mpeg-1/mp01-vid/index.htm.

[12] B. Gergel, H. Cheng, C. Nielsen, and X. Li, "A unified framework for image set compression." in *IPCV*, 2006, pp. 417–423.

[13] M. Stonebraker and G. Kemnitz, "The POSTGRES Next-Generation Database Management System," *Communications of the ACM*, vol. 34, no. 10, pp. 78–92, 1991.

[14] D. J. Abadi, S. R. Madden, and M. Ferreira, "Integrating compression and execution in column-oriented database systems," in *SIGMOD*, 2006, pp. 671–682.

[15] Z. Chen, J. Gehrke, and F. Korn, "Query optimization in compressed database systems," in *ACM SIGMOD*, 2001.

[16] G.Graefe and L.Shapiro, "Data compression and database performance," In ACM/IEEE-CS Symp. On Applied Computing pages 22 -27, April 1991.

[17] M. A. Roth and S. J. V. Horn, "Database compression," *SIGMOD Rec.*, vol. 22, no. 3, pp. 31–39, 1993.

[18] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte, "The implementation and performance of compressed databases," *SIGMOD Rec.*, vol. 29, no. 3, pp. 55–67, 2000.

APPENDIX

A. SciDB Versioning Examples

In this section, we illustrate how we are integrating our versioning system with the SciDB query language, AQL, through several language examples. Suppose we create a 3x3 SciDB array using the AQL command `CREATE UPDATEABLE ARRAY`:

```
CREATE UPDATABLE ARRAY Example
    ( A::INTEGER ) [ I=0:2, J=0:2 ];
```

`CREATE UPDATEABLE ARRAY` can create an array that contains multiple attributes (for example, `( A::INTEGER, B::DOUBLE )`), as well as arbitrarily many dimensions (for example, `[I=0:2, J=0:2, K=1:15, L=0:360]`). SciDB stores the newly-loaded data as one version of the array, as can be seen with the `VERSIONS` command:

```
VERSIONS(Example);
[('Example@1')]
```

Data is loaded into this array from disk via the `LOAD` command:

```
LOAD Example FROM 'array_file.dat';
```

Additional versions can subsequently be loaded:

```
LOAD Example FROM 'array_new_version.dat';
LOAD Example FROM 'array_another_new_version.dat';
VERSIONS(Example)
[('Example@1'),('Example@2'),('Example@3')]
```

Versions can then be retrieved by name and ID number, using the `SELECT` primitive; Support for selecting versions by conditionals or arbitrary labels is under development. For example, assuming that Example@3 was added on January 5, 2011 (and no other version was added on that date), it can be selected by date:

```
SELECT * FROM Example@'1-5-2011';
[
[(3),(6),(9)]
[(12),(15),(18)]
[(21),(24),(27)]
]
```

In some cases, it may be useful to select more than one version at a time. To support this case, a special form is introduced:

```
SELECT * FROM Example@*;
[
[(1),(2),(3)]
[(4),(5),(6)]
[(7),(8),(9)]
][
[(2),(4),(6)]
[(8),(10),(12)]
[(14),(16),(18)]
][
[(3),(6),(9)]
[(12),(15),(18)]
[(21),(24),(27)]
]
```

In this case, "Example@*" takes the two-dimensional "Example" array, and returns a three-dimensional array with all versions of "Example" lined up on the third axis. This form can be used with SciDB's `SUBSAMPLE` operator to allow the selection of a range of versions:

```
SELECT * FROM SUBSAMPLE
(Example@*, 0, 1, 1, 2, 2, 3);
[
[(8),(10)]
[(14),(16)]
][
[(12),(15)]
[(21),(24)]
]
```

This example selects coordinates 0 to 1 along the X axis, 1 to 2 along the Y axis, and 2 to 3 in the time dimension, returning a 2x2x2 array. "Example@*" is a first-class array object, so all other SciDB operators will work properly with it.

Arrays can also be branched. A branch is essentially a duplicate of an existing array, that can be updated separately. Branches are performed as follows:

```
BRANCH(Example@2 NewBranch);
LOAD NewBranch FROM 'other_version.dat';
LOAD NewBranch FROM 'another_version.dat';
```

Note that branches are formed off of a particular version of an existing array, not necessarily the most recent version, but they create a new array with a new name. This provides a means of adding data to a past version of an array.

B. Minimum Spanning Forest

We give below the algorithm for finding the minimum storage forest when materializing multiple versions may be beneficial.

```
/* Find Space Optimal Layout With One
   Materialized Version                          */
run Algorithm 1
/* Take Cheapest Materialization As Root         */
V^i_min = min(MM(i,i)) ∀ 1 ≤ i ≤ N
Λ = V^i_min
roots ⊕ V^i_min
/* Traverse MST and Add Deltas from Root         */
foreach Pair of Nodes (V^i, V^j) ∈ MST from root V^i_min do
    Λ = Λ ⊕ Δ^{i,j}
/* Consider Splitting the Graph by
   Materializing other Versions                  */
foreach version V^i | ∃MM(j,k) < MM(i,i) do
    Δ_diff = 0
    foreach version V^l on the path between V^i and roots do
        /* Find the Best Delta to Replace       */
        if Size(Δ_l) > MM(i,i) AND Size(Δ_l) > Δ_diff
        then
            Δ_diff = Size(Δ_l)
            V_ToReplace = V_l
    if Δ_diff > 0 then
        /* Split Graph and Add a New Root       */
        Λ = Λ ⊖ Δ_i
        Λ = Λ ⊕ V^l
        roots ⊕ V^l
store Λ
```

Algorithm 2: Minimum Spanning Forest